



CENTRE FOR
SYSTEMS
SCIENCE

LCCR

**Efficient Evaluation of Functional
Single Linear Recursions In
Deductive Databases**

Jiawei Han and Qiang Wang

CSS/LCCR TR 90-02

February 1990

Efficient Evaluation of Functional Single Linear Recursions in Deductive Databases[†]

Jiawei Han and Qiang Wang

School of Computing Science
Simon Fraser University
Burnaby, B.C., CANADA V5A 1S6
{han, qiang} @cs.sfu.ca

ABSTRACT

Functional single linear recursions are single linear recursions with function symbols. We study the compilation and efficient evaluation of functional single linear recursions in deductive databases and show that a functional single linear recursion can be rectified to a function-free single linear recursion, which can then be compiled to a highly regular form (a bounded, or an n -chain recursion). Compilation greatly facilitates the analysis of such recursions. By incorporating finiteness and monotonicity constraints, a large class of functional single linear recursions can be analyzed and evaluated efficiently in deductive databases.

1. Introduction

Function-free recursions, though representing an interesting class of logic programs, may have a relatively small set of database applications. Many deductive database problems, such as traversal recursions [RHD86], the manipulations of lists and complex data objects [Ull89], etc., can be represented by recursions with function symbols. It is important to study the efficient evaluation of such kind of recursions.

In this paper, we develop a technique for the compilation and efficient evaluation of *functional single linear recursions* (single linear recursions with function symbols). It consists of the following steps. First, a function-predicate transformation is performed, which maps a functional logic rule to its function-free counterpart. Then a transformed single linear recursion is compiled to a bounded, a single-chain, or a multi-chain recursion. When a query is posed to the system, the finite evaluability of the query on the compiled recursion is tested, its termination is justified, and an efficient evaluation plan is generated based on such an analysis.

Our study shows that the compilation benefits the analysis of functional linear recursions. Since the function-predicate transformation maps a functional recursion to a function-free one, many techniques developed for the compilation and evaluation of function-free linear recursions [BMS86, BaR86, BeR87, Han89b, HeN84, Ioa85, Sck89, Ull89] can be applied to functional linear recursions. Moreover, by incorporating finiteness and monotonicity constraints, a large class of functional linear recursions can be analyzed and evaluated efficiently.

[†] The work was supported in part by the Natural Sciences and Engineering Research Council of Canada under Grant A-3723 and a research grant from Centre for System Science of Simon Fraser University.

The paper is organized as follows. Section 2 studies the rectification and compilation of functional single linear recursions. Section 3 discusses the finite evaluability, termination and efficient evaluation of compiled recursions. Section 4 compares our method with other techniques. We conclude our discussion in section 5.

2. Compilation of Functional Single Linear Recursions

2.1. Functional single linear recursions

Like most researchers, we assume that a deductive database is partitioned into three portions: (i) an *extensional database (EDB)* (a set of extensional facts), (ii) an *intensional database (IDB)* (a set of intensional predicates defined by Horn-clause rules), and (iii) a set of *integrity constraints (ICs)*.

Definitions. A recursive rule is **linearly recursive** if the head predicate is the sole recursive predicate and appears exactly once in the body. A recursion is a **single linear recursion** if it consists of one linear recursive rule and one or more nonrecursive (**exit**) rules. A predicate logic is **function-free** if it does not contain function symbols; otherwise, it is **functional** [MaW88]. Similarly, we distinguish a **function-free Horn-clause** from a **functional Horn-clause** and a **function-free (single) linear recursion** from a **functional (single) linear recursion**.

Example 1. We present two typical functional single linear recursions, *flight* and *append*, as below. The notation used here is similar to Datalog [Ull89].

- (1a) $flight(Dep, Arr, Fare) :- edb\_flight(Dep, Arr, Fare).$
- (1b) $flight(Dep, Arr, Fare) :- flight(Dep, Int, F_1), edb\_flight(Int, Arr, F_2), Fare = F_1 + F_2.$
- (2a) $append([], L, L).$
- (2b) $append([X | L_1], L_2, [X | L_3]) :- append(L_1, L_2, L_3).$

The rules (1a) and (1b) define an IDB predicate $flight(Dep, Arr, Fare)$ which represents a connected flight with departure city *Dep*, arrival city *Arr* and fare *Fare*, where *Fare* is calculated by an arithmetic function which sums up the fares of the two connecting flights, that is, $Fare = F_1 + F_2$. The rules (2a) and (2b) define an IDB predicate *append* which concatenates two lists, where $[X | L_1]$ denotes a list construction function, $cons(X, L_1)$, which takes an element *X* as the head and a list *L₁* as the rest of the resulting list. $\square$

2.2. Rectification of functional logical rules

To facilitate the analysis of a functional recursion, a *function-predicate transformation* is performed which maps a function together with its *functional variable* to a predicate (called *functional predicate*), where the functional variable of a function is the variable which unifies/holds the returned value(s) of the function or the object identity if the function represents an object structure. That is, each function of arity *n* is transformed to a predicate of arity *n* + 1, with the last argument (in our convention) representing the unifying variable. The mapping is performed recursively when there are nested functions. To make explicit the linkage between a function and its functional predicate, we use the name of the function to represent its corresponding functional predicate. For example, $V = f(X_1, \dots, X_k)$ is transformed to $f(X_1, \dots, X_k, V)$. If there is a predicate in the original program carrying the same name but with one more argument than the function, we can easily rename the transformed functional predicate to avoid ambiguity. A similar transformation is also discussed in [KRS88, RBS87].

Since the transformation maps a functional logical rule to a function-free one, the compilation and analysis of a functional recursion can be performed in the framework of a function-free recursion, and the latter has been studied extensively in deductive database research. However, the characteristics of the functions should not be lost by the transformation. They should be registered as integrity constraints and used in the analysis of the transformed recursions, which will be discussed in the next section. A functional predicate can also be transformed back to its functional form, when necessary, which is called the *predicate-function transformation*.

As discussed in [Ull89], logical rules in different forms should be rectified to facilitate the compilation and analysis. The rules for predicate p is *rectified* if all the functions are mapped to the corresponding functional predicates by the function-predicate transformation, and all the heads of the rules are identical and of the form $p(X_1, \dots, X_k)$ for distinct variables $X_1, \dots, X_k$.

The rectification of a set of functional rules is performed by (i) the function-predicate transformation and (ii) the rectification of the transformed function-free rules. The former is stated above, while the latter has been studied in [Ull89], which introduces new variables for constants and repetitive variables in the head predicate and adds built-in predicates into the body to enforce whatever constraints the head predicate formerly enforced through constants and repetitions of variables. We examine some examples.

Example 2. The following rule set defines a functional single linear recursion in which $f(X)$ and $g(Y)$ are two unary functions.

(3a) $s(X, Y) :- a(X, Y).$

(3b) $s(X, Y) :- s(f(X), g(Y)).$

The recursion can be considered as a functional version of the same generation example [BMS86] by viewing $a(X, Y)$ as *sibling*(X, Y), and f and g as *father*/*mother*, or *parent* if we allow multiple-valued functions.

The rectification process transforms (3b) to

(3b') $s(X, Y) :- s(X_1, Y_1), f(X, X_1), g(Y, Y_1).$

where $X_1 = f(X)$ is mapped to $f(X, X_1)$ and $Y_1 = g(Y)$ is mapped to $g(Y, Y_1)$.

Similarly, *flight* and *append* of Ex. 1 can be rectified as follows, where *sum* represents the built-in function "+", and *cons* the list construction function.

(1a') $flight(D, A, F) :- edb\_flight(D, A, F).$

(1b') $flight(D, A, F) :- flight(D, I, F_1), edb\_flight(I, A, F_2), sum(F_1, F_2, F).$

(2a') $append(U, V, W) :- U = [], V = W.$

(2b') $append(U, V, W) :- append(U_1, V, W_1), cons(X_1, U_1, U), cons(X_1, W_1, W). \quad \square$

2.3. Compilation of functional single linear recursions

Function-free single linear recursions can be compiled into highly regular chain forms or bounded recursions [Han89a, YHH88]. Obviously, a functional linear recursion, after being transformed into its function-free counterpart, can be so compiled as well. Here we briefly outline the compilation process and refer a detailed discussion to [Han89a].

Definitions. In the compilation of a single linear recursion with the recursive predicate r , we consider that the first expansion of r is the recursive rule r itself. The k -th expansion of r ($k >$

1) is the unification of the recursive rule of r with the $(k - 1)$ -th expansion of r . The **0-th expanded exit rule set** is the set of nonrecursive rules of r . The **k-th expanded exit rule set** is generated by the unifications of the set of nonrecursive rules of r on the k -th expansion of r . The **compiled formula** of r is the set of formulas generated by all the expanded exit rules of r .

In the compilation of a functional single-linear recursion, functional predicates are treated as EDB predicates. However, since a functional predicate may represent an infinite relation, special care should be taken in the query analysis. For example, a set of *connected* (i.e., variable-shared) EDB predicates can be merged into one in compilation. However, the merged processing of some functional predicates may result in infinite relations. In this case, it is still necessary to separate them in the query processing. This will be discussed in the next section.

Example 3. We examine the compilation of the recursions s , $flight$ and $append$. First, the expansions of s lead to the compiled formula:

$$(3c) \quad s(X, Y) = a(X, Y) \cup \bigcup_{k=1}^{\infty} (f^k(X, X_1), g^k(Y, Y_1), a(X_1, Y_1)).$$

where the notation f^k represents a sequence of k f predicates, that is, f^1 is f itself, and for $k > 1$,

$$f^k(X, X_1) = f(X, T_1), f(T_1, T_2), \dots, f(T_{k-1}, X_1).$$

It can also be written as

$$f^k(X, X_1) = \begin{cases} f(X, X_1) & \text{if } k = 1, \\ f^{k-1}(X, T_{k-1}), f(T_{k-1}, X_1) & \text{if } k > 1. \end{cases}$$

The compiled formula (3c) indicates that the recursion consists of two length-equivalent chains, *f-chain* and *g-chain*, respectively.

The recursive rule $flight$ contains two connected EDB predicates, edb_flight and sum , which are merged into one in compilation, that is,

$$(1b'') \quad flight(D, A, F) :- flight(D, I, F_1), es(I, A, F_1, F).$$

where

$$es(I, A, F_1, F) :- edb\_flight(I, A, F_2), sum(F_1, F_2, F).$$

Thus the compiled formula of $flight$ consists of one chain,

$$(1c) \quad flight(D, A, F) = edb\_flight(D, A, F) \cup \bigcup_{k=1}^{\infty} (edb\_flight(D, I, F_1), es^k(I, A, F_1, F)).$$

where

$$es^k(I, A, F_1, F) = \begin{cases} es(I, A, F_1, F) & \text{if } k = 1, \\ es^{k-1}(I, T, F_1, F_k), es(T, A, F_k, F) & \text{if } k > 1. \end{cases}$$

The recursive rule $append$ contains two connected functional predicates, $cons$, which should be merged into one in compilation, that is,

$$(2b'') \quad append(U, V, W) :- append(U_1, V, W_1), cc(U_1, W_1, U, W).$$

where

$$cc(U_1, W_1, U, W) :- cons(X_1, U_1, U), cons(X_1, W_1, W).$$

Thus the compiled formula of $append$ consists of one chain as well,

$$(2c) \text{ append}(U, V, W) = (U = [], W = V) \cup \bigcup_{k=1}^{\infty} (U_k = [], W_k = V, cc^k(U_k, W_k, U, W)).$$

where

$$cc^k(U_k, W_k, U, W) = \begin{cases} cc(U_1, W_1, U, W) & \text{if } k = 1, \\ cc^{k-1}(U_k, W_k, U_1, W_1), cc(U_1, W_1, U, W) & \text{if } k > 1. \end{cases}$$

□

Based on this example, we formally define a chain and an n-chain recursion.

Definitions. A chain of length k ($k > 1$) is a sequence of k predicates with the following properties: (1) all k predicates have the same name, say P , and the l -th P of the chain is denoted as $P_{(l)}$, (2) there is at least one shared variable in every two consecutive predicates, and if i is the variable position in the first predicate, j the variable position in the second, and the two positions contain shared variables, then (i, j) is an invariant of the chain, i.e., the i -th variable of $P_{(l)}$ is shared with the j -th variable of $P_{(l+1)}$ for every l where $1 \leq l \leq k-1$. Each predicate of the chain is called a **chain element**. A chain element may also be formed by a sequence of *connected* nonrecursive predicates, which is called a **chain generating path**. A 0-length chain is defined as a tautology.

Definitions. A linear recursion is an **n-chain recursion** if there exist two positive integers c and l , and for any k -th expansion ($k > c \times l$), the compiled formula can be represented in an n-chain form, that is, one chain (when $n = 1$) or n synchronous (of the same length) chains (when $n > 1$) each with the length greater than l , and possibly some other predicates which do not form a chain. It is a **single-chain recursion** when $n = 1$, or **multi-chain recursion** when $n > 1$. A recursion is **bounded** if it is equivalent to a set of nonrecursive rules.

Single linear recursions can be classified into several classes based on the types of variable connections in the predicates [Han89a, YHH88]. The compilation of recursions in each class derives compiled formulas with similar characteristics. A detailed study of such a compilation process is in [Han89a], which derives the following theorem.

Theorem 1. A single linear recursion can be compiled to either an n-chain recursion or a bounded recursion.

The recursive rule of an n-chain recursion is called an **n-chain recursive rule**. An important property of an n-chain recursive rule is that it generates the same set of chain-elements with the same kind of variable connections at each expansion. A linear recursive rule may need to expand k times to become an n-chain rule. If so, the first k expanded exit rules, that is, from the 0-th to the $(k-1)$ -th expanded exit rules, are taken as the set of exit rules in the n-chain recursion. We examine one example.

Example 4. Suppose the variables X and Y are twisted in the body of the recursive rule s in Ex. 2 as shown below.

$$(4a) \quad s(X, Y) :- a(X, Y).$$

$$(4b) \quad s(X, Y) :- s(f(Y), g(X)).$$

The rectification process transforms (4b) into

$$(4b') \quad s(X, Y) :- s(Y_1, X_1), f(Y, Y_1), g(X, X_1).$$

Its first expanded exit rule is (4a') and the second expanded rule is (4b''), (or in the functional form (4b''')), which is a double-chain recursive rule (generating an *fg-chain* and a *gf-*

chain).

$$(4a') \quad s(X, Y) :- a(Y_1, X_1), f(Y, Y_1), g(X, X_1).$$

$$(4b'') \quad s(X, Y) :- s(X_2, Y_2), g(X, X_1), f(X_1, X_2), f(Y, Y_1), g(Y_1, Y_2).$$

$$(4b''') \quad s(X, Y) :- s(f(g(X)), g(f(Y))).$$

Thus the recursion is transformed into a double-chain recursion which consists of one double-chain recursive rule (4b'') and two exit rules, (4a') and (4a'). $\square$

2.4. Compilation benefits the analysis of functional linear recursions

Functional single linear recursions can be compiled to highly regular formulas by rectification and compilation. Such regularity greatly facilitates the analysis of such recursions.

First, compilation leads to the recognition of three basic categories: *bounded recursions*, *single-chain recursions* and *multi-chain recursions*. Different algorithms should be applied to different categories: *for bounded recursion, nonrecursive query processing algorithms are adequate; for single-chain recursions, transitive closure algorithms should be applied in general; while for multi-chain recursions, more general linear recursive query processing techniques, such as the counting method and the magic sets method [BMS86] should be considered.*

Moreover, compilation recognizes whether a function should be processed iteratively in a recursion. If there is no chain generating path in a recursion (i.e., a bounded recursion) or there is no function along a chain generating path (i.e., no function being part of a chain element), the recursion can be processed in a way similar to a set of nonrecursive rules or a function-free recursion. We examine two such examples.

Example 5. The following rule set is a bounded recursion with function symbols.

$$(5a) \quad r(X, Y) :- r(Y, X), a(X, f(Y)).$$

$$(5b) \quad r(X, Y) :- b(g(Y), h(X)).$$

The rectified recursive rule of (5a) is (5a'),

$$(5a') \quad r(X, Y) :- r(Y, X), a(X, W), f(Y, W).$$

It is a bounded recursion. This is because the second expanded exit rule is (5a'') which is absorbed by the tautological rule, " $r(X, Y) :- r(X, Y)$," and cannot generate any new results.

$$(5a'') \quad r(X, Y) :- r(X, Y), a(X, W), f(Y, W), a(Y, W), f(X, W).$$

Thus the recursion is equivalent to two nonrecursive rules, the exit rule and the first expanded exit rule, and the compiled formula is (by the predicate-function transformation):

$$(5c) \quad r(X, Y) = (b(g(X), h(Y)), a(X, f(Y))) \cup b(g(Y), h(X)).$$

Clearly, such a recursion can be processed by the query evaluation techniques for nonrecursive rules with function symbols [Zan85]. $\square$

Therefore, we have

Corollary 1. *A bounded recursion with function symbols can be evaluated by nonrecursive query evaluation methods.*

Example 6. The following rule set is a single-chain recursion with no function along the chain generating path.

$$(6a) \quad r(X, Y) :- a(X, W), r(W, Y), b(f(Y)).$$

$$(6b) \quad r(X, Y) :- c(X, Y).$$

This can be seen by rectifying the recursive rule (6a) to (6a'),

$$(6a') \quad r(X, Y) :- a(X, W), r(W, Y), b(Z), f(Y, Z).$$

The functional predicate, $f(Y, Z)$, is not connected to a chain generating path $a(X, W)$. Thus the function itself will not generate a chain in compilation. This can be seen by one more expansion on the recursive rule, which generates

$$(6a'') \quad r(X, Y) :- a(X, W), a(W, W_1), r(W_1, Y), b(Z_1), f(Y, Z_1), b(Z), f(Y, Z).$$

Notice that the expression " $b(Z), f(Y, Z), b(Z_1), f(Y, Z_1)$ " is equivalent to " $b(Z), f(Y, Z)$ " in (6a'') because neither Z nor Z_1 appears in the head predicate. Thus the compiled formula of the recursion is (by the predicate-function transformation),

$$(6c) \quad r(X, Y) = c(X, Y) \cup \bigcup_{k=1}^{\infty} (a^k(X, W), c(W, Y), b(f(Y))).$$

Clearly, there is no iterative processing on the function $b(f(Y))$ in the evaluation of the recursion. $\square$

Therefore, we have

Corollary 2. *There is no recursive or iterative processing on the functions which are not along the chain generating path in an n-chain recursion.*

3. Evaluation of Functional Single Linear Recursions

Now we study the evaluation of those n-chain recursions ($n \geq 1$) whose chain generating paths contain function symbols. Notice that the built-in predicates, such as $X < Y$, and the functional predicates whose corresponding functions are defined on infinite domains represent infinite relations. To ensure that an evaluation strategy computes all the answers and terminates properly, two issues should be examined: (1) *finite evaluability*, that is, the evaluation is performed on finite relations and generates finite intermediate relations at each iteration, and (2) *termination*, that is, the evaluation generates all the answers and terminates at finite number of iterations. The study of these issues needs the examination of two kinds of integrity constraints, *finiteness constraints* and *monotonicity constraints* [KRS88].

3.1. Finiteness constraints and finite evaluability

First, we introduce the concept of finiteness constraint and examine the finite evaluability of functional n-chain recursions.

Definition. A *finiteness constraint* (FC) over a predicate r is of the form $X \rightarrow Y$ where X and Y are sets of arguments (more exactly, argument positions). An instance of r satisfies this constraint if and only if for each tuple t in r , the set of tuples $\{s[Y] \mid s[X] = t[X]\}$ is finite.

In other words, a finiteness constraint $X \rightarrow Y$ over a predicate r implies that each value of attribute X corresponds to a finite set of Y values in r . Finiteness constraint is strictly weaker than the functional dependency studied in database theory [Ull89]. It holds trivially for all finite predicates. Since all the EDB relations are finite, all the arguments in EDB relations satisfy the finiteness constraint. In a functional predicate $f(X_1, \dots, X_n, V)$, if all the domains for arguments $X_1, \dots, X_n$ are finite, V must be finite no matter whether f is a single- or a multiple-valued function. Thus we have the finiteness constraint, $(X_1, \dots, X_n) \rightarrow V$. Most built-in predicates do

not have such a property. For example, it is impossible to assert that Y is finite from the built-in predicate $X > Y$ and a finite set of X . However, it is possible to do so if the built-in predicate is $X = Y$.

There are some common finiteness constraints for many functions. In many cases, one argument of a function can be computed from the values of the other arguments and the value of the function. For example, in the functional predicate $sum(X, Y, Z)$, any argument can be finitely computed if the other two arguments are finite. Such a relationship can be represented by a set of finiteness constraints, such as $(X, Z) \rightarrow Y$, and $(Y, Z) \rightarrow X$. However, not all the functions have such a property. For example, in the function $round(X)$ which returns a round-up integer value from a rational number X , it is impossible to assert the finiteness of X given $round(X) = 5$.

Specific finiteness constraints should be explored for specific functions. An interesting finiteness constraint, $Z \rightarrow (X, Y)$, holds in the functional predicate $cons(X, Y, Z)$, which indicates if the list Z is finite, there are only a finite number of choices (actually only one here) of X and Y .

Definition. A query is **finitely evaluable** if the evaluation of the query is performed on finite relations and produces finite intermediate relations (i.e., produces a finite relation at each iteration if the query is recursive).

Since a set of query constants may bind some infinite domain(s) to a finite set of arguments, the analysis of finite evaluability should incorporate query instantiation information. Similar to the notations used in the magic sets transformation [BaR86, Ull89], we use a superscript b or f to adorn a variable to indicate the variable being *bound (finite)* or *free (infinite)*, and use a string of b 's and f 's to adorn a predicate to indicate the bindings of its corresponding arguments.

All the variables in an EDB predicate or being equivalent to a set of constants are finite. Finiteness is propagated through finiteness constraints and variable equivalence. That is, from X^b, Y^b , and the finiteness constraint $(X, Y) \rightarrow Z$, we obtain Z^b ; and from (i) X^b and (ii) $X = U$ or $U = X$, we obtain U^b . We notice that for an n -chain recursion, the following two properties hold [Han89a]: (1) each expansion adds the same set of predicates to the body of the current recursive or expanded rule, and (2) for each expansion, the corresponding variable connections inside the chain elements and between the recursive predicates in the head and the body remain the same. Hence, we can easily derive an algorithm to test the finite evaluability in an n -chain recursion.

Algorithm 1. Testing the finite evaluability of a query in an n -chain recursion.

Input. (1) The n -chain recursive rule and the set of exit rules of an n -chain recursion, (2) a set of finiteness constraints, and (3) query instantiation information.

Output. An assertion of whether the query is finitely evaluable.

Method.

1. Test the finite evaluability of (1) the exit rule set, and (2) the first expanded exit rule by pushing the query instantiation information into the rules being tested and propagating the finiteness bindings based on (i) the query instantiations, (ii) the instantiations in the rule being tested, (iii) variable equivalence, and (iv) the set of finiteness constraints.
2. Return "yes" if every variable in the two rules being tested is finite after the finiteness binding propagation or "no" otherwise. $\square$

Theorem 2. Algorithm 1 correctly tests the finite evaluability of an n -chain recursion in $O(n)$ time, where n is the number of predicates in the recursion.

Proof sketch.

Obviously, the algorithm correctly tests the finite evaluability of the exit rule set and the first expanded exit rule themselves because a rule is not finitely evaluable if and only if there exists at least one variable which is not bound after the propagation. In the testing of the first expanded exit rule, the propagation passes the bindings through the chain elements. Suppose in rule (7), the i -th argument of the recursive predicate r in the body, X_i' , is finite and it passes the bindings through the chain element a_i to make the i -th argument in the head predicate, X_i , finite. Since the finiteness binding propagation at each expansion goes through the same set of chain elements and with the same set of variable bindings, a finite X_i' will make X_i bound in the next expansion. The same is true if the binding propagation is from X_i to X_i' .

$$(7) \quad r(\dots, X_i, \dots) :- a_i(\dots, X_i, \dots, X_i', \dots), \dots, r(\dots, X_i', \dots), \dots$$

Therefore, if all the variables are finitely evaluable, we can assert that the further expansions on the recursive rule are also finitely evaluable.

Since the test traverses each variable in the predicates of the rules being tested only once and the arity of each predicate is finite in the rules being tested, the algorithm terminates at $O(n)$ time, where n is the number of predicates in the rules being tested, or the number of predicates in the recursion. $\square$

Example 7. The recursion $flight(D, A, F)$ is finitely evaluable for any query. This is because (i) all the variables in the exit rule are finitely evaluable because they are in an EDB predicate; and (ii) all the variables in its first expanded exit rule are finite after the finiteness binding propagation. The initial bindings of the head predicate should be $flight(D^x, A^x, F^x)$, where the superscript x means either b or f (i.e., any query). After the finiteness binding propagation, the rule is shown in (8).

$$(8) \quad flight(D^b, A^b, F^b) :- edb\_flight(D^b, I^b, F^b), edb\_flight(I^b, A^b, F^b), sum(F^b, F^b, F^b).$$

The binding propagation proceeds as follows: (i) D , I and F_1 are finite because they are in an EDB predicate, and also I , A and F_2 ; and (ii) F is finite because both F_1 and F_2 are finite and there is a finiteness constraint, $(F_1, F_2) \rightarrow F$, in sum . Since the finiteness of every variable is not dependent on query bindings, the recursion is finitely evaluable for any query.

The query, " $?-append([a, b], V, W)$.", is not finitely evaluable because there remain variables adorned with f in both the exit rule and the first expanded exit rule after the binding propagation. We show the testing of the first expanded exit rule. Initially, the head predicate is $append(U^b, V^f, W^f)$, after the binding propagation, the adorned rule is (9a).

$$(9a) \quad append(U^b, V^f, W^f) :- U^b = [], V^f = W^f, cons(X^b, U^b, U^b), cons(X^b, W^f, W^f).$$

The binding propagation proceeds as follows: (i) U_1 and U are adorned with b from the instantiations in the exit rule and the query; and (ii) X_1 is adorned with b from the finiteness constraint, $U \rightarrow (X_1, U_1)$, in $cons(X_1, U_1, U)$. However, V , W and W_1 remain to be adorned with f . Thus the query is not finitely evaluable.

However, the query, " $?-append(U, V, [a, b, c])$.", is finitely evaluable because all the variables in the two test rules are adorned with b after the binding propagation. Initially, the head predicate is $append(U^f, V^f, W^b)$, after the binding propagation, the adorned first

expanded exit rule is (9a').

(9a') $append(U^b, V^b, W^b) :- U^b = [], V^b = W^b, cons(X^b, U^b, U^b), cons(X^b, W^b, W^b).$

The binding propagation proceeds as follows: (i) U_1 and W are adorned with b from the instantiations in the exit rule and the query; (ii) X_1 and W_1 are adorned with b from the finiteness constraint, $W \rightarrow (X_1, W_1)$, in $cons(X_1, W_1, W)$, (iii) V is finitely evaluable based on the variable equivalence $V = W_1$, and finally, (iv) U is adorned with b from the finiteness constraint, $(X_1, U_1) \rightarrow U$, in $cons(X_1, U_1, U)$. Therefore, the query is finitely evaluable. $\square$

Notice that the direction of the finiteness binding propagation dictates the order of the query evaluation. If there are several choices in the propagation of the finiteness bindings, one can select a preferred order for efficient query evaluation. This will be discussed in Section 3.5.

3.2. Monotonicity constraints

A nonrecursive query, if finitely evaluable, will terminate and derive a finite set of answers. However, a finitely evaluable recursive query guarantees only a finite intermediate relation at each iteration but not the termination of the iterative processing. Monotonicity constraints are often useful for the termination of the recursive/iterative processing.

Interestingly, a query always terminates on the recursions in which all the functions are defined on finite domains because the transformed functional predicates can be viewed as finite EDB relations, on which a query terminates using the well-developed function-free recursive query evaluation techniques [BaR86]. For example, if both f and g in the recursion s of Ex. 2 are defined on finite domains, such as *father* of a real database, the evaluation of the recursion terminates. Thus we have,

Theorem 3. *If all the predicates and functions of a functional n -chain recursion are defined on finite domains, the recursion is finitely evaluable and terminating for any query.*

Proof sketch.

If a predicate or a function is defined on finite domain(s), it can be mapped to a finite EDB relation. Since a function-free recursive query on finite EDB relations is finitely evaluable and terminating based on the study of function-free recursive query processing [BaR86], we have the conclusion. $\square$

However, query processing may not terminate on the recursion in which there exists a function defined on infinite domains and along a chain generating path. For example, *Fare* of the predicate *flight* is defined on infinite domains. The evaluation of a query, " $? - flight(boston, chicago, F)$," may not terminate if there is no restraint on the possible values of *Fare* and the $Dep \rightarrow Arr$ pairs in the EDB form cycles. Monotonicity constraints often play an important role in the termination of the evaluation of such functional recursions.

Definition. A monotonicity constraint (MC) is a relationship $r_i \Rightarrow r_j$, ($\Rightarrow$: a partial order), where r_i and r_j are two arguments of a predicate r . The constraint holds in an instance of r if and only if the value in the i -th argument is strictly greater than that in the j -th argument in every tuple, according to some partial order $\Rightarrow$.

There are typical monotonic behaviors in arithmetic functions, list functions and term construction functions. Such behaviors should be specified as monotonicity constraints and used in query evaluations. Also, an acyclic EDB relation is a partially ordered finite relation which is useful in the judgement of termination.

Example 8. We examine some monotonic behaviors in arithmetic functions, list functions and term construction functions.

1. An arithmetic operation often implies the monotonicity of a function. For example, $F_1 > 0$, $F_2 > 0$, and $F_1 + F_2 = F$ implies that $F > F_1$ and $F > F_2$.
2. The monotonicity of a list manipulation function usually lies at the growing or shrinking of the length of the list. For example, a list construction operation, such as *CONS*, results in a longer list than the original ones, while some other list operations, such as *CAR* or *CDR*, result in an atom or a list with the length shorter than the original one.
3. The monotonicity of a term constructor/de-constructor is similar to that of a list operation. The repetitive application of a term constructor results in an increasingly deeply-nested sequence of functors, such as $f(f(\dots f(X)\dots))$, while the repetitive application of a term de-constructor, such as f^{-1} , results in an increasingly less-nested sequence of functors.
4. One may like to terminate a recursion after a certain number of iterations when there is no appropriate termination condition. It essentially treats the number of iterations as a monotonically increasing function and implies an arithmetic monotonicity constraint. $\square$

3.3. Termination of query evaluation in single-chain recursions

The processing of a single-chain recursion is essentially the iterative processing of a growing sequence of chain elements. An argument of the head predicate is *monotonic* if the argument has certain monotonic behavior as the number of iteration increases. A query terminates if there exists a *termination restraint* which blocks the growth of the monotonic argument. A termination restraint can be provided by a query, by a rule, or implied in an EDB relation. Termination of a query should be justified before the query processing starts. We examine two examples.

Example 9. A termination restraint can be specified or implied in a query or a rule. In Ex. 1, the value of F in the head predicate *flight* is monotonically increasing because the MC, $F > F_1$, in the recursive rule (1b') indicates that F value increases at each iteration. Thus a restraint on the maximum value of F , such as $F < 1000$, in the recursive rule or in the query, ensures the termination of the query evaluation.

Similarly, the lengths of U and W in the head predicate *append* (Ex. 1) are monotonic functions because of the MC's, $\text{length}(U) > \text{length}(U_1)$ and $\text{length}(W) > \text{length}(W_1)$ in (2b'). Thus when the query is finitely evaluable, an instantiation of U or W in the query ensures the termination of the query evaluation because the iteration terminates when a list shrinks down to an empty list. $\square$

Example 10. A query evaluation can also be terminated by a termination restraint implied in an EDB relation or by the existence checking evaluation strategy if the query is an existence checking query [Han89b]. Suppose the recursion is defined below (from [Ull89]),

- (10a) $p(X) :- p(f(X)).$
 (10b) $p(X) :- a(X).$

Since (10a'), the rectified rule of (10a), is a single-chain recursive rule with the chain element $f(X, X_1)$, its expansions generate a sequence of increasingly nested *f-functors* on X .

- (10a') $p(X) :- f(X, X_1), p(X_1).$

Suppose there is only one fact, $a(f(f(1)))$, in the EDB. The maximum nesting level for the functor f in the EDB is two, which can be used as a termination restraint. For example, the query, " $?-p(2)$," terminates at the third iteration with the answer, "no", because $a(2)$, $a(f(2))$, and $a(f(f(2)))$ do not satisfy the EDB and further iterations generate deeper nested f 's than any one in the EDB. Similarly, the query, " $?-p(X)$," terminates at the third iteration with the answer set, $X = \{f(f(1)), f(1), 1\}$.

Some queries can also be evaluated by an existence checking evaluation. For example, the query, " $?-p(f(1))$," terminates at the second iteration because once the query is satisfied, it is unnecessary to perform further evaluations.

Notice that any query on the above recursion will terminate if the EDB is finite. This is because the growth of the monotonic argument must match some EDB facts in the exit rule. However, if the growth of the argument were in the other direction (i.e., starting at the EDB facts), a termination restraint should be specified at the other end of the chain instead of implied by the EDB. For example, if the recursive rule is $(10a'')$ instead of $(10a)$,

$(10a'') \quad p(f(X)) :- p(X).$

the query, " $?-p(X)$," can never be terminated by the EDB. A termination restraint should be specified in the query to block the infinite growth of the f -functors. $\square$

In summary, we have

Theorem 4. *The evaluation of a single-chain recursion terminates if (1) there exists a restraint in a query, in a rule, or in an EDB, which blocks the growth of a monotonic argument in the direction of its growth; or (2) the query is an existence checking query and it meets the existence checking condition.*

Proof sketch.

For (1), since the argument is monotonic and there exists a restraint which blocks the growth of the argument in the direction of its growth, the query must terminate at certain iteration. For (2), based on the nature of the existence checking query [Han89b], the query terminates since it is unnecessary to continue searching once the search meets the existence checking condition. $\square$

3.4. Termination of query evaluation in multi-chain recursions

Multi-chain recursions share some commonalities with single-chain recursions but also raise some new issues on the termination of query evaluations. First, a termination restraint provided by a query, a rule, or an EDB can be used to block a monotonic argument of any one chain in the direction of its growth. Since the processing of one chain must pass the exit portion (the body of the exit rule) in order to synchronize other chains, the finiteness of the EDB or other restraints in the exit rule can often be used in the judgement of termination. Moreover, since all the chains must be synchronized in a multi-chain recursion, the termination of one chain finishes the processing of the other synchronous chains.

Example 11. If the functional predicate f of the recursion s in Ex. 2 corresponds to an acyclic EDB relation. The query, " $?-s(1, Y)$," terminates because the processing of the f -chain cannot proceed after a finite number of iterations, which terminates the evaluation of the other chain as well. On the other hand, if f is a term constructor, the recursion generates a monotonically increasingly nested f -functors. The maximum nesting level of the f -functor in the EDB relation of the exit rule terminates the evaluation. $\square$

In a multi-chain recursion, different chains may have different monotonic behaviors, and their interactions via the exit portion may substantially complicate their combined behaviors. If the combined effect of the synchronized chains still has some monotonic behavior, the restraints studied above are still useful to terminate the recursion.

Example 12. Suppose the query " $?-s(1, Y)$." is posed on the recursion below, where f is a functor.

(11a) $s(X, Y) :- X = Y.$

(11b) $s(X, Y) :- s(f(X), f(Y)).$

The combined effect of the two synchronized chains is to make the value stable (that is, passing one f , one equivalence, and then one f^{-1}). Thus the query returns $Y = 1$ and terminates. Without knowing this behavior, the recursion will derive an infinite number of 1's without termination.

Suppose that the recursive rule is (11b') instead of (11b),

(11b') $s(X, f(Y)) :- s(f(X), Y).$

the combined effect of the two chains is that Y in $s(X, Y)$ corresponds to X or a monotonically increasing doubly nested f 's of X . Thus the answer to the query " $?-s(1, Y)$." should be $Y = \{1, f(f(1)), f(f(f(f(1))))\dots\}$, which can never terminate if there is no restraint specified in the query. On the contrary, if Y is given, such as, " $?-s(X, f(f(2)))$.", the query terminates with the answer $X = \{f(f(2)), 2\}$. $\square$

In summary, we have,

Theorem 5. *The evaluation of a functional multi-chain recursion terminates if (1) the evaluation of any one chain terminates; (2) a restraint provided by a query, a rule, or an EDB blocks the growth of a monotonic argument of any one chain in the direction of its growth; (3) the combined effect of the synchronized chain is monotonic on an argument, and there is a restraint which blocks the growth of the argument in the direction of its growth; or (4) the query is an existence checking query and it meets the existence checking condition.*

Proof sketch.

For (1), the processing terminates because the termination of any one chain evaluation at its k -th expansion makes the other chains not synchronizable beyond the k -th expansion. Conditions (2), (3) and (4) are similar to Theorem 4. $\square$

3.5. Query evaluation methods

A query evaluation plan should be generated based on the analysis of the finite evaluability and the termination of a query. A function defined on finite domains can be treated as an EDB relation in the analysis of a query. However, a function defined on infinite domains cannot be evaluated until the bindings of its arguments make it finitely evaluable. Therefore, special care should be taken in the evaluation of those chains whose functions are not immediately finitely evaluable. Especially, a query may make some functions in a chain immediately finitely evaluable but make other functions in the same chain element evaluable only when the bindings in the exit portion has been set up. Such a chain should be split in its evaluation to make the evaluable portion proceed and the non-evaluable portion buffered. This can be seen from the analysis of some queries on *append*.

Example 13. We study the query evaluations on *append*(U, V, W). First, we examine the

evaluation of the query, " $?-append([a, b], V, [a, c, d]).$ ", whose adorned predicate is $append^{fb}$. The application of Algorithm 1 asserts the finite evaluability of the query. Since both monotonic arguments U and W are instantiated, the recursion terminates. There are two possible finiteness binding propagations among predicates in the body of the first expanded exit rule. One is similar to the next query example, which requires the splitting of a chain in the evaluation. The other does not require a chain-splitting evaluation, which is stated as follows. Starting at the query instantiations U^b and W^b and the exit rule binding U^f , the finiteness bindings are propagated by variable equivalence and finiteness constraints as shown in Fig. 1. That is, from W to W_1 and X_1 , from U to X_1 and U_1 , and then from W_1 to V .

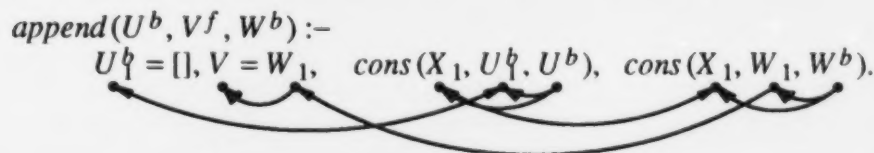


Figure 1. Binding propagations for " $?-append(U^b, V^f, W^b).$ ".

We take the first $cons$ as c_1 , the second one as c_2 , and the exit portion, " $(U_1 = [], V = W_1)$ ", as e . Both c_1 and c_2 are finitely evaluable without going through the exit portion. Thus c_1 and c_2 can be evaluated together using the provided query constants. For the query " $?-append([a, b], V, [a, c, d]).$ ", it proceeds as follows: (i) evaluate the 0-th expanded exit rule, e , which derives no answer because $U = [a, b]$ which does not match $[]$; (ii) evaluate the first expanded exit rule, that is, c_1, c_2 and then e , which derives $X_1 = a, U_1 = [b]$, and $W_1 = [c, d]$, but no answer to the query because U_1 does not match the body of the exit rule; (iii) evaluate the second expanded exit rule, that is, using the bindings produced in the evaluation of c_1 and c_2 at the last iteration, evaluate c_1, c_2 and then e_2 . However, since $U_1 = [b]$ and $W_1 = [c, d]$ cannot produce a consistent X_2 , the unification fails, and the processing cannot proceed. The query terminates with an answer "no". This can be verified by Prolog execution.

Secondly, we examine the evaluation of another query " $?-append(U, V, [a, b]).$ ", whose adorned predicate is $append^{fb}$. Similar to the first query, it is finitely evaluable and terminating. Its detailed binding propagation in the test rule is shown in Fig. 2. That is, from W to W_1 and X_1 , then to V in the exit portion, and from the exit portion U_1 and X_1 (passed from c_2) to U . Since the monotonic argument W is instantiated (bound), the recursion terminates.

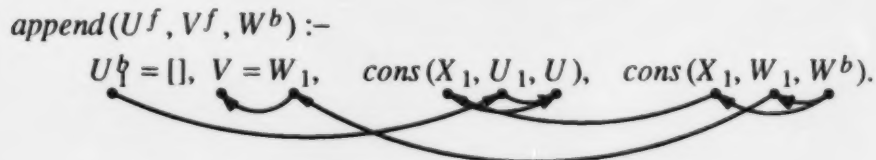


Figure 2. Binding propagations for " $?-append(U^f, V^f, W^b).$ ".

Obviously, c_1 and c_2 are not finitely evaluable by starting at the exit portion. The only direction which are finitely evaluable is c_2, e and then c_1 . The evaluation should proceed as follows: (i) evaluate the 0-th expanded exit rule, e , which derives $U = []$ and $V = [a, b]$ (the first set of the answer to the query); (ii) evaluate the first expanded exit rule in the order of c_2, e , and c_1 , which derives $X_1 = a, V = W_1 = [b], U_1 = []$, and $U = [a]$ (i.e., the second set of the answer is, $U = [a]$, and $V = [b]$); and (iii) evaluate the second expanded exit rule in the order of c_2, c_2, e, c_1 , and c_1 , while the first c_2 is evaluated previously which can be used in this evaluation (an

idea from Henschen and Naqvi [HeN84]), that is, it starts at $W_1 = [b]$, which derives $X_2 = b$, $V = W_2 = []$, $U_2 = []$, and $U = [a, b]$ by processing two c_1 's (i.e., the third set of the answer is, $U = [a, b]$, and $V = []$). The further iteration cannot proceed since $W_2 = []$. Thus the processing terminates with the same set of answers as the Prolog execution. $\square$

The evaluation of the second query of Ex. 13 is similar to that of a double-chain recursion except the X_1 value of each c_2 is buffered and then passed to the corresponding X_1 of c_1 . The processing splits one chain into more than one based on the finite evaluability of functions. Then it adopts the technique of processing a multi-chain recursion except the values of the connecting variables are buffered in the processing of one "chain" and passed to the corresponding portions of the other "chain(s)". We call such a technique the *chain-splitting processing technique*.

The above analysis shows that besides using the well-developed recursive query evaluation techniques, such as performing selection first [BaR86, HeN84], magic sets and counting [BMS86], etc., the query evaluation of a functional n-chain recursion must follow the order of the propagation of the finiteness bindings. Otherwise, the evaluation will produce infinite intermediate relations. To enforce such an order, the chain-splitting processing technique should be adopted when necessary.

4. A Comparison with Other Relevant Methods

Safetiness, termination and efficient evaluation of functional recursions have been investigated in previous studies [BrS89, RBS87, SaZ86, SaV89, Ull89], which paves the way for our study. Sagiv and Vardi [SaV89] studied the safety of Datalog queries over infinite relations and viewed safety as a combination of two properties: *weak safety*, which guarantees the finiteness of intermediate answers, and *termination*, which guarantees the finiteness of the evaluation. Their study shows that weak safety can be tested in polynomial time if there is a bound on the arity of the IDB predicate. The weak safety corresponds to the finite evaluability of our study. We developed a linear algorithm for testing the finite evaluability of an n-chain recursion, which is consistent with their result on the computational complexity. Furthermore, [SaV89] showed that the judgment of termination is a difficult problem. Our study provides only a set of sufficient conditions for terminating a functional n-chain recursion, which are nevertheless useful in many practical applications. Our study is confined to single linear recursions. More work is needed to extend the results to general functional recursions.

Many previous studies suggest the application of the generalized magic sets method in the evaluation of functional linear recursions [Ull89]. Thus it is important to compare the generalized magic sets method with ours. Our technique includes the magic sets method as one evaluation technique for compiled multi-chain recursions. However, the selection of an evaluation method is determined by the compilation and analysis of a functional single linear recursion. The functions which are in a bounded recursion or not along a chain generating path can be evaluated using nonrecursive or function-free recursive query processing techniques. Moreover, based on the analysis of one chain or a small number of synchronous chains, finite evaluability and termination can be justified and efficient evaluation plans can be generated for a large class of compiled n-chain recursions. However, the magic sets method does not compile a single linear recursion into chain forms, which cannot take advantages of the well-formed chains in the analysis of the behaviors of single linear recursions.

Furthermore, the generalized magic sets method may meet challenges in the evaluation of some functional linear recursions involving finite evaluability and termination problem. Here we present one example which shows the nontermination problem in the derivation of magic sets.

Example 14. For the recursion presented in Ex. 10, the magic set for the query, " $?-p(1)$," should be derived using the following set of rewritten rules (see [Ull89]):

(12a) $m\_p(1)$.

(12b) $m\_p(f(X)) :- m\_p(X)$.

which results in $\{1, f(1), f(f(1)), f(f(f(1))), \dots\}$, an infinite magic set. However, using our method as stated in Ex. 10, the evaluation terminates with the answer "yes". The termination is justified by the existence checking evaluation or by the monotonicity constraint plus the termination restraint, the maximum nesting level of the functor f in the EDB. $\square$

Finally, in comparison with PROLOG's query evaluation method, our method is set-oriented, independent of the order of the rules and the order of the predicates in the rules, and easy to generate sub-optimal query processing plans by incorporating quantitative measurements and search heuristics, such as *performing selection first*, etc. Such arguments have been well supported by many studies in deductive databases [BaR86, BeR87, Ull89].

5. Conclusions

We studied the compilation and efficient evaluation of functional single linear recursions in deductive databases. A method is developed which rectifies a functional single linear recursion to a function-free single linear recursion, compiles it to a bounded or an n -chain recursion, and then analyzes its finite evaluability and termination based the compiled form, query instantiations, and the finiteness and monotonicity constraints. Queries which are finitely evaluable and terminating can be evaluated efficiently under the guidance of the finiteness binding propagation and using the techniques similar to the evaluation of function-free linear recursions.

The functional recursions studied in this paper are confined to single linear recursions. The compilation and efficient evaluation of the functional recursions which consist of multiple linear recursive rules and/or nonlinear recursive rules are interesting topics for the future research.

References

- [BMS86] F. BANCILHON, D. MAIER, Y. SAGIV and J. D. ULLMAN, Magic Sets and Other Strange Ways to Implement Logic Programs, *Proceedings of 5th ACM Symposium on Principles of Database Systems*, Cambridge, MA, March 1986, 1-15.
- [BaR86] F. BANCILHON and R. RAMAKRISHNAN, An Amateur's Introduction to Recursive Query Processing Strategies, *Proceedings of 1986 ACM-SIGMOD Conference on Management of Data*, Washington, DC, May 1986, 16-52.
- [BeR87] C. BEERI and R. RAMAKRISHNAN, On the Power of Magic, *Proceedings of the 6th ACM Symposium on Principles of Database Systems*, San Diego, CA, March 1987, 269-283.
- [BrS89] A. BRODSKY and Y. SAGIV, On Termination of Datalog Programs, *Proceedings of the 1st International Conference on Deductive and Object-Oriented Databases (DOOD'89)*, Kyoto, Japan, December 1989, 95-112.
- [Han89a] J. HAN, Compiling General Linear Recursions by Variable Connection Graph Analysis, *Computational Intelligence*, 5(1), 1989, 12-31.
- [Han89b] J. HAN, Multi-Way Counting Method, *Information Systems*, 14(3), 1989, 219-229.

- [HeN84] L. J. HENSCHEN and S. NAQVI, On Compiling Queries in Recursive First-Order Databases, *Journal of ACM*, 31(1), 1984, 47-85.
- [Ioa85] Y. E. IOANNIDIS, A Time Bound on the Materialization of Some Recursively Defined Views, *Proceedings of the 11th International Conference on Very Large Data Bases*, Stockholm, Sweden, August 1985, 219-226.
- [KRS88] R. KRISHNAMURTHY, R. RAMAKRISHNAN and O. SHMUELI, A Framework for Testing Safety and Effective Computability of Extended Datalog, *Proceedings of the 7th ACM Symposium on Principles of Database Systems*, Austin, TX, March 1988, 154-163.
- [MaW88] D. MAIER and D. S. WARREN, *Computing with Logic: Logic Programming with Prolog*, Benjamin/Cummings, 1988.
- [RBS87] R. RAMAKRISHNAN, F. BANCILHON and A. SILBERSCHATZ, Safety of Recursive Horn Clauses with Infinite Relations, *Proceedings of the 6th ACM Symposium on Principles of Database Systems*, San Diego, CA, March 1987, 328-339.
- [RHD86] A. ROSENTHAL, S. HEILER, U. DAYAL and F. MANOLA, Traversal Recursion: A Practical Approach to Supporting Recursive Applications, *Proceedings of 1986 ACM-SIGMOD Conference on Management of Data*, Washington, DC, May 1986.
- [SaZ86] D. SACCA and C. ZANIOLO, The Generalized Counting Method for Recursive Queries, *Proceedings of 1st International Conference on Database Theory*, Rome, Italy, 1986, 31-53.
- [SaV89] Y. SAGIV and M. VARDI, Safety of Datalog Queries over Infinite Databases, *Proceedings of the 8th ACM Symposium on Principles of Database Systems*, Philadelphia, PA, March 1989, 160-171.
- [Sek89] H. SEKI, On the Power of Alexander Templets, *Proceedings of 8th ACM Symposium on Principles of Database Systems*, Philadelphia, PA, March 1989, 150-159.
- [Ull89] J. D. ULLMAN, *Principles of Database and Knowledge-Base Systems*, Vols. 1 & 2, Computer Science Press, 1989.
- [YHH88] C. YOUN, L. J. HENSCHEN and J. HAN, Classification of Recursive Formulas in Deductive Databases, *Proceedings of 1988 ACM-SIGMOD Conference on Management of Data*, Chicago, IL, June 1988, 320-328.
- [Zan85] C. ZANIOLO, The Representation and Deductive Retrieval of Complex Objects, *Proceedings of the 11th International Conference on Very Large Data Bases*, Stockholm, Sweden, 1985, 458-469.

Centre for Systems Science

- Simon Fraser University established the **Centre for Systems Science** as a provincial centre for advanced study. The objective of CSS is twofold: coordinate and combine advanced systems research at SFU, and encourage application of research results within industry in order to further the technological development of British Columbia.

- CSS has assembled three research groups to streamline its efforts. The orientation of these groups is as follows: intelligent systems, microelectronics, and computer and communications systems. Because of shared research interests, this latter group merges with LCCR (**Laboratory for Computer and Communications Research**) for the publication of technical reports.

- CSS's three groups involve over 80 skilled faculty researchers principally from SFU's Schools of Engineering Science and Computing Science, but also draw substantially on Mathematics, Physics, Geography, Philosophy, Linguistics, Psychology, Communication, and Kinesiology.